A machine-checked soundness proof for an efficient verification condition generator

Frédéric Vogels
Katholieke Universiteit Leuven
Celestijnenlaan 200a
3001 Heverlee, Belgium
+32 16 328289
frederic.vogels@cs.kuleuven.be

Bart Jacobs
Katholieke Universiteit Leuven
Celestijnenlaan 200a
3001 Heverlee, Belgium
+32 16 327826
bart.jacobs@cs.kuleuven.be

Frank Piessens
Katholieke Universiteit Leuven
Celestijnenlaan 200a
3001 Heverlee, Belgium
+32 16 327603
frank.piessens@cs.kuleuven.be

ABSTRACT

Verification conditions (VCs) are logical formulae whose validity implies the correctness of a program with respect to a specification. The technique of checking software properties by specifying them in a program logic, then generating VCs, and finally feeding these VCs to a theorem prover, is several decades old. It is the underlying technology for state-of-the-art program verifiers such as the Spec$^\sharp$ programming system, or ESC/Java. The classic way of computing VCs is by means of Dijkstra's weakest precondition calculus. However, modern verification condition generators (VCgens), including Spec$^\sharp$ and ESC/Java's VCgens, are based on an optimized version of this algorithm, that avoids an exponential growth of the VCs in the length of the program to be verified. For this optimized VCgen algorithm, only informal soundness arguments are available. The main contribution of this paper is a fully formal, machine-checked proof of the soundness of such an efficient VCgen algorithm.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, Correctness proofs, Programming by contract, Formal methods, Validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions*; F.4.1 [**Logics and Meanings of Programs**]: Mathematical Logic—*Mechanical theorem proving*

General Terms

Algorithms, Security, Languages, Verification

Keywords

Soundness proof, verification conditions

1. INTRODUCTION

25th Annual ACM Symposium on Applied Computing Software Verification and Testing Track 2010, Sierre, Switzerland

The generation of a verification condition (VC) for a given program is a common technique used by verification tools to check the correctness of the program. This VC is then fed into an interactive proof assistant (e.g. Coq or Isabelle) or, preferably, an automated SMT solver (e.g. Z3 or Simplify) to determine its validity, which would imply the correctness of the program. ESC/Modula-3 [16], ESC/Java [5, 9, 13], Boogie [1, 19, 2, 17, 15, 20] and Why [8, 18, 7] are examples of tools using this technique.

A VC has many different but mathematically equivalent formulations, and the choice of which formulation one uses can influence the performance of the verification process dramatically. For example, the translation using weakest preconditions as described in [6] produces VCs which grow exponentially in the size of the program. In order to make verification practically possible it is important to avoid this exponential blowup.

Whereas the classic computation of VCs has been studied thoroughly, and even a machine-checked proof of soundness of this VC generation algorithm is available [24], this is not the case for the more efficient algorithms.

The contribution of this paper is a machine-checked proof using Coq [3] for the VC generation algorithm (without support for raise/catch) described by Leino [14], which is essentially a reformulation of the algorithm by Flanagan and Saxe [10]. Leino defines a "dream property" which allows one to rewrite the verification condition in such a way that its size grows polynomially (instead of exponentially) in program size. The dream property is only true for so-called passive commands, i.e. commands which have no net effect on the program state. It is however possible to define a program transformation – passification – which removes all non-passive commands such as assignments and replaces them by something equivalent [10].

The generation of VCs is done in multiple steps. We start with an overview (Section 2) after which we zoom in on each one separately and discuss them more thoroughly in Sections 3, 4, 5 and 6. All theorems have been proved in Coq: the script can be found online at [4]. Throughout the text we refer to Coq definitions using this font.

Due to space restrictions, we do not provide proofs but instead refer the interested reader to [23, 4].

2. OVERVIEW

The approach we discuss to generate verification conditions comprises a number of phases as shown in Figure 1. The source code to be verified is first translated into an in-

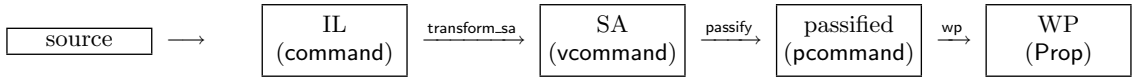

Figure 1: Overview

termediate language (IL), which we formally define in Section 3. An IL is used for the same reasons compilers do: the same back-end can be reused for many different languages. In our case this means the same technology can be used for verifying C, Java, C♯, ... [19, 2, 17, 15, 20]. The translation from source code to IL falls outside the scope of this paper. Such translations (including machine checked proofs) have been done before for Java bytecode [12] and for a small OO source language [24].

In our Coq proof, IL programs are represented by the command inductive data type, and their semantics are defined by the step relation. This part is in fact the Achilles heel of the script: an error could invalidate all proofs. Fortunately, this "axiomatic part" amounts to only about 150 lines, most of which is trivial. The main complexity lies in the definition of the operational semantics, which is why we included a number of theorems showing that the described behaviour does conform to our expectations.

Secondly, the IL program is transformed into single assignment form (SA), which is the subject of Section 4. The main theorem of that section is sa_transformation_fail, which states the original IL-program and its SA-form behave "similarly". Approximately 1500 lines of the Coq script are dedicated to this phase. SA-programs are represented by a different type in the Coq script, namely vcommand, for reasons made clear later. The reduction rules carry the name vstep and behave in the exact same way as step.

Next comes passification, another program transformation which is detailed in Section 5. Analogously with the previous transformation, the main goal of that section is to show that the SA-program and its passified form behave similarly (vmultistep_pmultistep_fail). Definitions and proofs related to this phase take up about 1200 lines of the script. Passified programs have their own specialized type pcommand, and predictably, a pstep relation describes their behaviour.

The last step is to generate the efficient weakest precondition from the passified program. In Section 6, we prove their soundness (soundness_efficient_wp) and show their size is polynomial with respect to the size of the original IL-code (polynomial_wps). This part amounts to around 1000 lines.

3. INTERMEDIATE LANGUAGE

The IL is no ordinary programming language: instead of representing a computation, it is meant to encode execution paths populated with assertions, i.e. conditions that are expected to evaluate to true, regardless of the path through which it was reached. The language provides a small number of commands (command), listed in Figure 2. We define the operational semantics as a binary relation between states, called the single step relation (step) and written $\sigma_1 \longrightarrow \sigma_2$. We distinguish two kinds of state (state): in-progress states consisting of a command and a store (denoted $\langle c, \mu \rangle$), and failure states consisting of just a store (failure μ). A store is a total mapping from identifiers to values. The single step relation is defined as the smallest set of state pairs obeying the rules appearing in Figure 3.

We summarily explain the commands' behaviour:

- **assert** e demands that e evaluates to true in the current store, or failure ensues. This command can be used to prevent division by zero errors or null dereferences, or to enforce programmer-specified correctness conditions.
- **assume** e can be used to prune unimportant paths from the execution tree. For example, the Java programming language guarantees the `this` pointer can never be `null`; the **assume** command can then be used to inform the verifier it should not consider execution paths where `this` is bound to `null`.
- **skip** is a no-operation, defined to simplify the operational semantics.
- $x := e$ binds x to the value e evaluates to in the current store. This command allows us to easily model execution paths of stateful programs, but becomes a burden when it comes to producing weakest preconditions.
- $c_1; c_2$ provides command sequencing.
- $c_1 \,[\!]\, c_2$ is the sole source of nondeterminism: execution can proceed with either c_1 or c_2. This command makes it possible to model conditionals and loops.

We also define a multiple step relation (multistep) in the obvious way.

Note that the operational semantics for this IL is nondeterministic (see the reduction rules for $c_1 \,[\!]\, c_2$): the execution of a program can be seen as a tree whose root is the initial state, whose leaves represent stuck states[1] and where nodes with multiple branches correspond to a nondeterministic choice between possible execution paths. One of our goals is to prevent the occurrence of failure states in this tree. Weakest preconditions express the minimal requirements on the initial store for this to be the case.

We would like to highlight a peculiarity of the language: since the store is a total mapping, there is no such thing as an "empty initial store". This means variables need no declaration prior to use, but also leads to the question, what value do unassigned variables have? The answer is simple: a variable is bound to whatever value the store associates it with (i.e. there are no default values nor is there a special "uninitialized" value). Then what store should be used at the beginning of execution? For the purpose of verification, all possible stores must be considered. So, not only do we have to deal with entire execution trees instead of linear execution paths, but also with an infinitude of such trees. None of these trees are allowed to have a failure state in them.

[1]We define a stuck state as one for which no reduction rule applies. For example, failure states are stuck states (but not vice versa!) Section 6 discusses stuck states elaborately.

$$c \;::=\; \textbf{skip} \mid \textbf{assert}\ e \mid \textbf{assume}\ e \mid x := e \mid c_1; c_2 \mid c_1 [\!] c_2$$

```
Inductive command : Set :=
| cSkip     :                          command
| cAssert   : expr                  -> command
| cAssume   : expr                  -> command
| cAssign   : id -> expr            -> command
| cSequence : command -> command -> command
| cChoice   : command -> command -> command.
```

Figure 2: Intermediate language grammar

$$\frac{e(\mu) = \textbf{true}}{\langle \textbf{assert}\ e, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu\rangle} \qquad \frac{e(\mu) \neq \textbf{true}}{\langle \textbf{assert}\ e, \mu\rangle \longrightarrow \mathsf{failure}\ \mu}$$

$$\frac{e(\mu) = \textbf{true}}{\langle \textbf{assume}\ e, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu\rangle} \qquad \frac{\langle c_1, \mu\rangle \longrightarrow \langle c_1', \mu'\rangle}{\langle c_1; c_2, \mu\rangle \longrightarrow \langle c_1'; c_2, \mu'\rangle}$$

$$\frac{\langle c_1, \mu\rangle \longrightarrow \mathsf{failure}\ \mu'}{\langle c_1; c_2, \mu\rangle \longrightarrow \mathsf{failure}\ \mu'} \qquad \frac{}{\langle \textbf{skip}; c_2, \mu\rangle \longrightarrow \langle c_2, \mu\rangle}$$

$$\frac{}{\langle x := e, \mu\rangle \longrightarrow \langle \textbf{skip}, \mu[x \mapsto e(\mu)]\rangle}$$

$$\frac{}{\langle c_1 [\!] c_2, \mu\rangle \longrightarrow \langle c_1, \mu\rangle} \qquad \frac{}{\langle c_1 [\!] c_2, \mu\rangle \longrightarrow \langle c_2, \mu\rangle}$$

Figure 3: Single step reduction rules

4. SINGLE ASSIGNMENT FORM

This section describes an algorithm to transform an arbitrary command to single assignment form (SA), a form where each variable is assigned to at most once during execution *and is not read from prior to this assignment.* As noted in the previous section, all variables have values pre-assigned to them. For example, the program **assert** $x = 3; x := 8$ assigns a value to x twice: the implicit initial binding, and the assignment of 8 to x. Thus, the example program is *not* SA. A SA-transformation would be **assert** $x = 3; y := 8$.

The algorithm presented in this paper extends identifiers with version numbers to deal with this issue in a simple way. One can imagine all identifiers in the original program have an implicit version number of 0 associated with them, e.g. **assert** $x_0 = 3; x_0 := 8$. Transforming this into SA then becomes a matter of simply incrementing the version number in each assignment: **assert** $x_0 = 3; x_1 := 8$. Similarly, all expressions need to be updated so they refer to the correct version of variables. Thus, using version numbers is an easy way to achieve "variable freshness."

To keep track of variable versions, we use a version map (vmap), which is a total map from identifiers to natural numbers. The transformation algorithm (transform_sa, Figure 4) then takes a command and version map, and returns a transformed command and a new version map, e.g.

$$\text{transform\_sa}(\textbf{assert}\ x = 3; x := 8, \lambda\, id\,.\,0) = (\textbf{assert}\ x_0 = 3; x_1 := 8, (\lambda\, id\,.\,0)[x \mapsto 1])$$

A separate id type (vid), command type (vcommand) and operational semantics (vstep, vmultistep) had to be defined in the Coq script to deal with the version extensions. These are virtually identical to their nonversioned twins, with the exception that all identifiers carry numbers with them.

The SA transformation is rather straightforward except for choice commands. It would be tempting to turn the program graph into a tree (e.g. changing $(c_a [\!] c_b); c_2$ into $(c_a; c_2) [\!] (c_b; c_2)$), which leads to a series of deterministic programs which can be dealt with separately in a simple manner. However, the number of such programs grows exponentially with the number of choice commands, which conflicts with the original goal of avoiding exponential blowup during VC generation.

A better, more efficient way to deal with choice commands consists of first transforming both branches to SA separately. Since both branches could perform assignments to different variables, their versions could get "out of sync": the algorithm returns different version maps for either branch. Consider the following example:

$$x := 0; y := 0; (x := 5 [\!] y := 5); \textbf{assert}\ x \neq y$$

A first attempt to transform this to SA could be

$$x_1 := 0; y_1 := 0; (x_2 := 5 [\!] y_2 := 5); \textbf{assert}\ x_2 \neq y_2$$

This is clearly not correct: if the left path is taken, x_2 should be compared to y_1, and conversely, if the right path is taken, x_1 should be compared to y_2.

To solve this problem, we build a "target version map" ν_t which both branches have to accommodate to: at the end of both their executions, all variables should have the versions mentioned in ν_t. In the case of our example, with $\nu_t = (\lambda\, id\,.\,0)[x \mapsto 2][y \mapsto 2]$, we get

$$\ldots; (x_2 := 5; y_2 := y_1 [\!] y_2 := 5; x_2 := x_1); \textbf{assert}\ x_2 \neq y_2$$

The $y_2 := y_1$ and $x_2 := x_1$ commands can be seen as synchronization commands, which transform the current version map to the target version map.

In order to transform $c_a [\!] c_b$ with initial version map ν_0, the actual algorithm proceeds as follows: it first transforms both branches c_a and c_b normally, resulting in transformed commands c_a' and c_b' and updated version maps ν_a and ν_b. It then builds (join) a target version map by taking the maximum version number for each variable: $\nu_t(x) = \max(\nu_a(x), \nu_b(x))$. Next, it creates (sync_vcommand) synchronization commands d_1 and d_2 to handle the version transition from ν_a and ν_b to ν_t. The final result of the transformation of the choice command is then $(c_1'; d_1 [\!] c_2'; d_2, \nu_t)$. Since there are an infinite number of identifiers, sync_vcommand can't just scan every identifier in search of differing version numbers to generate appropriate assignments for them; instead, it needs to have an idea (more precisely, a finite idea) of which identifiers could have different version numbers. This problem is solved by collecting all assignment targets (targets) of both branches into a finite set and passing it along to sync_vcommand.

Definition 1. We say stores are synchronized with respect to a version map ν (store_sync_vstore) when

$$\mu \sim^{\nu} \mu_{\text{v}} \equiv \forall\, x \bullet \mu(x) = \mu_{\text{v}}(x_{\nu(x)})$$

Theorem 1 (sa_transformation_fail) states that if the original program fails (i.e. there exists an execution path leading to a failure state), so will its SA transformation.

```
Fixpoint transform_sa c v :=
 match c with
 | cAssert e => (vcAssert (version_expr e v), v)
 | cAssume e => (vcAssume (version_expr e v), v)
 | cAssign x e =>
    (vcAssign (x, 1 + v x) (version_expr e v),
     inc v x)
 | cSequence c1 c2 =>
    let (c1', v')  := transform_sa c1 v in
    let (c2', v'') := transform_sa c2 v' in
     (vcSequence c1' c2', v'')
 | cSkip => (vcSkip, v)
 | cChoice c1 c2 =>
    let (c1', v1') := transform_sa c1 v in
    let (c2', v2') := transform_sa c2 v in
    let t1 := targets c1 in
    let t2 := targets c2 in
    let vt := join v1' v2' in
    let t  := IdSet.union t1 t2 in
    let d1 := sync_vcommand t v1' vt in
    let d2 := sync_vcommand t v2' vt in
      (vcChoice (vcSequence c1' d1)
                (vcSequence c2' d2), vt)
 end.
```

Figure 4: SA Algorithm

LEMMA 1. *(sa_transformation_skip) If*

$$\langle c, \mu\rangle \longrightarrow^* \langle \mathbf{skip}, \mu'\rangle \quad \wedge \quad \mu \sim^{\nu} \mu_{\mathrm{sa}}$$

then, with $(c', \nu') = \mathrm{transform\_sa}(c, \nu)$

$$\langle c', \mu_{\mathrm{sa}}\rangle \longrightarrow^* \langle \mathbf{skip}, \mu'_{\mathrm{sa}}\rangle \quad \wedge \quad \mu' \sim^{\nu'} \mu'_{\mathrm{sa}}$$

THEOREM 1. *With* $(c', \nu') = \mathrm{transform\_sa}(c, \nu)$,

$$\langle c, \mu\rangle \longrightarrow^* \mathsf{failure}\ \mu' \Rightarrow \mu \sim^{\nu} \mu_{\mathrm{sa}} \Rightarrow \langle c', \mu_{\mathrm{sa}}\rangle \longrightarrow^* \mathsf{failure}\ \mu'_{\mathrm{sa}}$$

5. PASSIFICATION

The SA transformation discussed in the previous section still produces assignments (in fact, it even adds some). In order to produce our efficient weakest preconditions, we need to get rid of those. This is where passification comes in: this second transformation rewrites all assignments as assumptions. The following example program clearly fails:

$$x := 1; x := x + 1; \mathbf{assert}\ x = 3$$

First it is transformed into SA:

$$x_1 := 1; x_2 := x_1 + 1; \mathbf{assert}\ x_2 = 3$$

Passification turns this into

$$\mathbf{assume}\ x_1 = 1; \mathbf{assume}\ x_2 = x_1 + 1; \mathbf{assert}\ x_2 = 3$$

Note the importance of the SA-transformation: without it we would get

$$\mathbf{assume}\ x = 1; \mathbf{assume}\ x = x + 1; \mathbf{assert}\ x = 3$$

which would not fail, contrary to the first three programs.

As in Section 4, we define a new command type (pcommand), which does not provide an assignment operation. A new set of reduction rules is also required (pstep, pmultistep), which lack assignment-describing logic. Since the store is never updated, we can factor it out: instead of $\langle c, \mu\rangle \longrightarrow \langle c', \mu\rangle$, we write $c \xrightarrow{\mu} c'$, and similarly for the multiple step relation.

```
Fixpoint passify c : pcommand :=
 match c with
 | vcAssert e => pcAssert e
 | vcAssume e => pcAssume e
 | vcSkip     => pcSkip
 | vcSequence c1 c2 =>
    pcSequence (passify c1) (passify c2)
 | vcChoice c1 c2 =>
    pcChoice (passify c1) (passify c2)
 | vcAssign x e =>
    assume_from_assign x e
 end.
```

Figure 5: Passification algorithm

The passification algorithm (passify), shown in Figure 5, is rather straightforward. We demonstrate its soundness by proving that if the execution of the SA-transformation of a given command encounters failure, so will its passification (Theorem 2, vmultistep_pmultistep_fail).

Definition 2. We define store equivalence up to a certain version map as

$$\mu \overset{\leq\nu}{\sim} \mu' \equiv \forall\, x, n \bullet n \leq \nu(x) \Rightarrow \mu(x_n) = \mu'(x_n)$$

LEMMA 2. *(vmultistep_pmultistep_skip) Let*

$$(c', \nu') = \mathrm{transform\_sa}(c, \nu)$$

then

$$\langle c', \mu\rangle \longrightarrow^* \langle \mathbf{skip}, \mu'\rangle \Rightarrow \mu' \overset{\leq\nu'}{\sim} \mu'' \Rightarrow c' \xrightarrow{\mu''}{}^* \mathbf{skip}$$

LEMMA 3. *(single_assignment_monotonic_store_fail) Let*

$$(c', \nu') = \mathrm{transform\_sa}(c, \nu)$$

then

$$\langle c', \mu\rangle \longrightarrow^* \mathsf{failure}\ \mu' \Rightarrow \mu \overset{\leq\nu}{\sim} \mu'$$

THEOREM 2. *Let* $(c', \nu') = \mathrm{transform\_sa}(c, \nu)$, *then*

$$\langle c', \mu\rangle \longrightarrow^* \mathsf{failure}\ \mu'' \quad \Rightarrow \quad \mathrm{passify}(c') \xrightarrow{\mu''}{}^* \mathsf{failure}$$

6. WEAKEST PRECONDITIONS

In this section we discuss the notion of weakest preconditions: we formally define them, prove their soundness, and show how their size varies in terms of program size.

When looking at the operational semantics reduction rules (Figure 3), we can identify three classes of stuck states:

- there is no way out of a failure state.
- **assume** commands can block further execution in case their associated expression does not evaluate to true in the current store. These may also be nested within a complicated sequencing structure (nested_assume).
- if execution can proceed unhindered, it will eventually end up with a **skip** command: this is as close as we can get to successful program termination.

```
Fixpoint wp vmu c Q {struct c} : Prop :=
 match c with
  | pcAssert e       => (e vmu = T) /\ Q
  | pcAssume e       => e vmu = T -> Q
  | pcChoice c1 c2   => wp vmu c1 Q /\
                        wp vmu c2 Q
  | pcSequence c1 c2 => wp vmu c1 (wp vmu c2 Q)
  | pcSkip           => Q
 end.

Fixpoint wlp vmu c Q {struct c} : Prop :=
 match c with
  | pcAssert e       => e vmu = T -> Q
  | pcAssume e       => e vmu = T -> Q
  | pcChoice c1 c2   => wlp vmu c1 Q /\
                        wlp vmu c2 Q
  | pcSequence c1 c2 => wlp vmu c1 (wlp vmu c2 Q)
  | pcSkip           => Q
 end.
```

Figure 6: Weakest preconditions

We define two kinds of weakest preconditions:

- the weakest *conservative* precondition (wp) of a program c with respect to a predicate Q, written $\mathbf{wp}(c, Q)$, is the condition which the initial store has to satisfy so that execution cannot reach failure, and that for every execution path ending up in **skip** (i.e. the third class of stuckness), the final store satisfies Q;
- the weakest *liberal* precondition (wlp) of a program c with respect to a predicate Q, denoted $\mathbf{wlp}(c, Q)$, allows failure to occur, and only guarantees that when execution reaches **skip**, the final store will satisfy Q.

One way of defining these [6] is shown in Figure 6. Theorem 3 states that given a passified program c, if the weakest conservative preconditions are true for some store μ, execution starting with this store will not reach failure.

THEOREM 3. *For any passified program c,*

$$\forall\, Q \;\bullet\; \mu \vDash \mathbf{wp}(c, Q) \;\Rightarrow\; \neg\, c \xrightarrow{\mu}{}^{*} \mathsf{failure}$$

We do not rely on the property that Q holds after successful execution, but $\mathbf{wp}(c, Q) = \mathbf{wp}(c; \mathbf{assert}\ Q, \mathbf{true})$ together with Theorem 3 and the operational semantics shows that Q holds. A similar case can be made for **wlp**.

The problem with the definitions given in Figure 6 is the size of the resulting VCs. The troublemaker is once more the choice command, which again leads to an exponential explosion, as the predicate Q is duplicated. However, it is possible to rewrite this so that Q appears only once [14, 10]. First, we split up the guarantees made by the weakest conservative preconditions into two cases: the program must not fail, and each time execution reaches **skip**, Q must be satisfied (wp_rewrite):

THEOREM 4. *For any passified program c,*

$$\mathbf{wp}(c, Q) \Leftrightarrow \mathbf{wp}(c, \mathbf{true}) \wedge \mathbf{wlp}(c, Q)$$

Next, we rewrite the weakest liberal preconditions as "either execution does not reach **skip**, or Q was true all along" (wlp_rewrite).

```
Fixpoint efficient_wlp vmu c Q : Prop :=
 match c with
 | pcAssert e => e vmu = T -> Q
 | pcAssume e => e vmu = T -> Q
 | pcSkip           => Q
 | pcSequence c1 c2 =>
    efficient_wlp vmu c1 (efficient_wlp vmu c2 Q)
 | pcChoice c1 c2   =>
    (efficient_wlp vmu c1 False /\
     efficient_wlp vmu c2 False) \/ Q
 end.

Fixpoint efficient_wp vmu c Q : Prop :=
 match c with
 | pcAssert e       => e vmu = T /\ Q
 | pcAssume e       => e vmu = T -> Q
 | pcSkip           => Q
 | pcSequence c1 c2 =>
    efficient_wp vmu c1 (efficient_wp vmu c2 Q)
 | pcChoice c1 c2   =>
    efficient_wp vmu c1 True /\
    efficient_wp vmu c2 True /\
    efficient_wlp vmu (pcChoice c1 c2) Q
 end.
```

Figure 7: Efficient weakest preconditions

THEOREM 5. *For any passified program c,*

$$\mathbf{wlp}(c, Q) \Leftrightarrow \mathbf{wlp}(c, \mathbf{false}) \vee Q$$

Note that these reformulations are only valid in a stateless context, hence the need for passification. This leads to the alternate formulation of the weakest preconditions shown in Figure 7. We are now ready to demonstrate the soundness of the efficient weakest preconditions (Theorem 6, soundness_efficient_wp).

THEOREM 6. *Let c_p be the passified SA-form of c*

$$\vDash \mathbf{wp}_e(c_p, \mathbf{true}) \Rightarrow \forall\, \mu\, \mu' \;\bullet\; \neg\, \langle c, \mu\rangle \longrightarrow^{*} \mathsf{failure}\ \mu'$$

Finally, we must prove our claim that the weakest preconditions are polynomial in size with respect to program size. This bound is *not* tight: proving the tight bound correct in Coq revealed to be quite difficult.

THEOREM 7. *Let c be a program, then*

$$|\mathbf{wp}_e(c, Q)| = O(|c|^4 + |Q|)$$

7. RELATED WORK

There is a huge amount of related work in program verification in general and hence, we necessarily focus on research results most closely related to ours. For a good overview of the state-of-the-art in program verification in general, we refer to [11].

A first related line of work is the research on verifiers that rely on VC generation, in particular on the optimized VC generation that is the subject of this paper. This includes the ESC/Modula-3 verifier [16], the ESC/Java verifiers [9, 13], the VCC verifier [19] and the Spec$^\sharp$ verifier [2].

The Why/Krakatoa/Caduceus line of tools [8] is a very interesting competitor to the Boogie/Spec$^\sharp$/VCC line of tools:

both toolsets are similarly built around an intermediate verification language and provide front-ends for Java-like and C-like languages. To avoid trust in the tool-chain, the Why VC generator adopts an approach where the proof produced, possibly with the help of other tools or the user, is checked a posteriori by an automatic checker.

Another important related area of research is the development of machine-checked proofs of programming language properties in general. The POPLmark challenge [22] is a set of benchmarks designed to evaluate the progress of mechanization of metatheory of programming languages, but the focus is more on type systems than on verification. In the Mobius project [21], machine-checked proofs of many programming language properties are being studied with the purpose of supporting proof-carrying code to certify security-related properties of programs. The project has worked out a machine-checked proof of the soundness of their program logic (the Mobius Base Logic) with respect to an operational semantics of Java bytecode. As part of the Mobius project, Lehner and Müller have shown the translation of Java bytecode to BoogiePL sound [12]. A machine-checked proof of the soundness of standard weakest-precondition based VC generation for a variant of the Boogie intermediate language is also available [24].

8. CONCLUSION

A chain is only as trustworthy as the weakest of its links. Verification has relied on automated checking by the computer, based on theoretical ideas which thus far were only proved informally. For example, the proofs in [10] which inspired this paper consist of just one line. Our contribution is to provide a Coq-certified single-assignment transformation and passification algorithm accompanied by a number of machine-checked soundness proofs, further guaranteeing correctness and thus giving us assurance that current and future verifiers built upon this technology can indeed be trusted. The full Coq proof script can be found online [4] and is included in [23].

9. REFERENCES

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005, volume 4111 of LNCS*, pages 364–387. Springer, 2006.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS 04*, pages 49–69. Springer, 2004.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[4] Coq script. `http://www.cs.kuleuven.be/~frederic/papers/efficient/passification.v`.

[5] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Report 159 extended static checking.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] J.-C. Filliâtre and C. Marché. Multi-prover Verification of C Programs. In *ICFEM*, pages 15–29, 2004.

[8] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV*, pages 173–177, 2007.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.

[10] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.

[11] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.

[12] H. Lehner and P. Müller. Formal Translation of Bytecode into BoogiePL. *Electr. Notes Theor. Comput. Sci.*, 190(1):35–50, 2007.

[13] K. R. M. Leino. Extended Static Checking: A Ten-Year Perspective. In *Informatics*, pages 157–175, 2001.

[14] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[15] K. R. M. Leino. Specification and verification of object-oriented software. Marktoberdorf lecture notes, 2008.

[16] K. R. M. Leino and G. Nelson. An Extended Static Checker for Modula-3. In *CC*, pages 302–305, 1998.

[17] K. R. M. Leino and W. Schulte. A verifying compiler for a multi-threaded object-oriented language. Marktoberdorf lecture notes, 2007. In Manfred Broy, Johannes Grünbauer, Tony Hoare (eds.). Software System Reliability and Security. IOS Press, 2007.

[18] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.

[19] W. Schulte, S. Xia, J. Smans, and F. Piessens. A Glimpse of a Verifying C Compiler — Extended Abstract, 2007.

[20] J. Smans, B. Jacobs, and F. Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *FMOODS '08: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, pages 220–239, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] The Mobius Project. `http://mobius.inria.fr`.

[22] The POPLmark Challenge. `http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=The_POPLmark_Challenge`.

[23] F. Vogels, B. Jacobs, and F. Piessens. A machine-checked soundness proof for an efficient verification condition generator: technical report. Technical Report CW568, Katholieke Universiteit Leuven, 2009.

[24] F. Vogels, B. Jacobs, and F. Piessens. A machine checked soundness proof for an intermediate verification language. In M. Nielsen, A. Kucera, P. B. Miltersen, C. Palamidessi, P. Tuma, and F. D. Valencia, editors, *35th International Conference on Current Trends in Theory and Practice of Computer Science (Sofsem 2009)*, volume 5404 of *Lecture Notes in Computer Science*, pages 570—581. Springer, 2009.